Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Tests de aleatoriedad para alfabetos arbitrarios

23 de Noviembre de 2021

Tesis de Licenciatura en Ciencias de la Computación

Nicolás Andrés Donatucci
LU 263/13
nadonatucci@gmail.com

Directores: Verónica Becher y Santiago Figueira

Resumen

Para medir calidad de la aleatoriedad de una muestra de datos se aplican tests estadísticos de aleatoriedad. La referencia clásica sigue siendo el conjunto de tests compilado por Donald Knuth en *The Art of Computing Programming*, Volumen 2, 1997. La implementación de código abierto y libre de algunos de estos tests y otros es la batería del *National Institute of Standards and Technology* (NIST) de Estados Unidos del año 2010, pero acepta solamente secuencias binarias, es decir, secuencias de ceros y unos. Si nuestro problema es medir aleatoriedad de secuencias de símbolos de un alfabeto más grande, la batería de tests de NIST no se puede aplicar, porque no hay ninguna manera de transformar las secuencias de símbolos de una alfabeto arbitrario a secuencias binarias, preservando la calidad de aleatoriedad.

En este trabajo hacemos una implementación de la batería de tests estadísticos de aleatoriedad de Knuth para secuencias de símbolos de alfabetos arbitrarios. Desarrollamos la batería en Python y está disponible para su uso libre. Damos explícitamente los parámetros para correr cada test.

Abstract

To measure the quality of randomness of a given data sample, one can apply statistical randomness tests. The classic reference for this, still, is the set of tests compiled by Donald Knuth in The Art of Computing Programming, Volume 2, 1997. The open source implementation of some of these tests plus others is the NIST (National Institute of Standards and Technology, USA) battery, but this one only accepts binary sequences (This means, sequences of zeroes and ones). If our problem consists in measuring the randomness of sequences of symbols of bigger alphabets, then the NIST battery can't be applied, given that there is no certainty that one can transform the sequences of symbols from one alphabet to binary sequences preserving the quality of randomness.

In this work we make an implementation of the randomness statistical tests proposed by Knuth. The battery is developed in Python and available for free use. We define the parameters to run each test.

Agradezco a Verónica Becher y Santiago Figueira por la guía, paciencia y calidez a lo largo de este proceso.

Agradezco a todos los docentes que ataqué a consultas sin piedad y que me ayudaron a entender, aprender, con amor por la materia y la enseñanza: Ariel Molinuevo, Maria Emilia Descotte, Agustín Gravano, Julieta Molina, Mariela Sued, Nicolás Rosner, Santiago Alvarez Colombo, Francisco Gomez Fernandez (Pachi), Paula Chocron, Isabel Mendez Díaz, Santiago Palladino, David Gonzalez Marquez, Alejandro Strejilevich, Melanie Sclar, Gonzalo Lera Romero, Manuel Gimenez, Esteban Lanzaroti, Viviana Cotik, Sergio Abriola, Pablo Brusco, Brian Cardiff.

Agradezco a mis compañeros de TP, que soportaron mis incertidumbres y nervios: Gianfranco Zamboni, Francisco Noriega, Gustavo Cairo, Nicolás Chamo, Ezequiel Zimenspitz (Especial mención porque sin tu insistencia no estaría acá)

Por último, agradezco a mis padres, Carlos Donatucci y Patricia Spagnolo por absolutamente todo, y a Elsa Estévez porque se ve que sus palabras algo lograron.

CONTENIDOS

1. ¿QUÉ HACEMOS EN ESTE TRABAJO?

Dado un alfabeto finito Σ, una secuencia de símbolos de dicho alfabeto se considera aleatoria si no hay evidencia de lo contrario. Existen tests de aleatoriedad que pueden ser utilizados para examinar la secuencia y mostrar evidencia en favor de la falta de aleatoriedad. Cada test está diseñado para analizar un aspecto particular que una secuencia aleatoria debería cumplir. Según el resultado de los tests, decimos que la secuencia es aleatoria con cierto nivel de confianza, o que no cumple alguna propiedad que debería, si lo fuera.

Implementaciones conocidas de software libre de baterías de tests de aleatoriedad son *Die Hard* [5] (y su versión mejorada DieHarder) y *NIST* [6]. Ambas asumen que el alfabeto del cual provienen las secuencias es $\Sigma = \{0, 1\}$.

Estas implementaciones resultan inadecuadas cuando se trabaja con conjuntos de datos que provienen de alfabetos con más de dos símbolos, como por ejemplo las letras del alfabeto, los números de la lotería, etc.

En este trabajo implementamos y adaptamos una batería de tests, aceptando secuencias que provengan de alfabetos de longitud arbitraria. También mostramos que no es fiable realizar transformaciones uno a uno de secuencias de alfabetos arbitrarios a secuencias del alfabeto $\Sigma = \{0, 1\}$ y evaluar éstas en su lugar.

La batería implementada consiste de un subconjunto de la batería de tests para alfabetos arbitrarios, tomada del volumen 2 del libro de Knuth [3] con algunas modificaciones propuestas por Koçak et al. [4].

Mostramos también los resultados de correr la batería sobre un conjunto de secuencias provenientes de un alfabeto de diez símbolos y luego transformándolas a secuencias provenientes de un alfabeto de dos símbolos.

2. TESTS DE ALEATORIEDAD

2.1. Marco teórico

Hablamos de *secuencias* aleatorias con una distribución específica. Esto significa que cada símbolo fue obtenido al azar, sin estar relacionado a otros de la secuencia, y que cada símbolo tiene una probabilidad específica de encontrarse en cualquier rango de valores. En una secuencia proveniente de un alfabeto de diez símbolos, donde los símbolos son equiprobables, cada símbolo debería aparecer $\frac{1}{10}$ de las veces, cada par de símbolos consecutivos $\frac{1}{100}$, y así sucesivamente. De todas formas, una secuencia verdaderamente aleatoria de un millón de símbolos no va a tener siempre 100000 apariciones de cada símbolo. De hecho, las chances son bajas. Una secuencia de estas secuencias va a tener esa característica en promedio.

Nuestro propósito es poder decidir si una secuencia es *suficientemente* aleatoria. La estadística nos provee de medidas de aleatoriedad. Los tests que pueden diseñarse son ilimitados. Si una secuencia se comporta de forma aleatoria con respecto a los tests $T_1 \dots T_n$ no podemos estar seguros de que no va a fallar cuando se le aplique el test T_{n+1}. Sin embargo, cada test nos da un poco mas de confianza en la aleatoriedad de la secuencia.

Los tests de aleatoriedad parten de una hipótesis, llamada *hipótesis nula*, usualmente notada como H_0, a la que se pone a prueba la secuencia a testear.

> H_0: La secuencia a testear fue generada por un mecanismo que produce números aleatoriamente, con igual probabilidad para cada uno de los símbolos del alfabeto y de forma independiente (conocer algunos resultados no da ninguna información sobre los restantes).

El paradigma de test de hipótesis provee herramientas potentes para refutar H_0 cuando la secuencia evaluada la contradice. Sin embargo no hay herramientas para confirmar la hipótesis. Las herramientas sólo pueden ofrecer respuestas del tipo "la secuencia a testear es compatible con H_0" o "la secuencia a testear no contradice H_0". La evidencia en contra de H_0 se cuantifica numéricamente con un número real entre 0 y 1 que se denomina p-valor. A grandes rasgos, los p-valores muy bajos son suficientes para refutar H_0, mientras que para validarla, es necesario realizar muchos tests independientes sobre distintas secuencias y testear la *meta-hipótesis*

> $\mathbb{H}_0$: : Los p-valores producidos por los tests están uniformemente distribuidos en el intervalo $[0, 1]$.

Test χ^2 (Chi cuadrado): Este test permite evaluar qué tan bien se ajustan n observaciones independientes de datos categóricos (con k categorías) a sus valores esperados, utilizando la suma de las diferencias cuadradas entre las observaciones y los datos esperados, pesándolas en base a la probabilidad de cada categoría. Para obtener el p-valor resultado se utiliza la distribución χ^2 con $k-1$ grados de libertad (esto se debe a que sabiendo la cantidad total de observaciones y la cantidad de observaciones que pertenecen a $k-1$ de las categorías, se puede deducir cuántas pertenecen a la categoría restante). Las secuencias generadas serán las observaciones, comparándose

con los resultados que se esperarían si éstas fueran aleatorias. Valores muy bajos indican evidencia en contra de la aleatoriedad de la secuencia.

Test de Kolmogorov-Smirnov: Este test permite decidir si un conjunto de números reales generados en forma independiente están distribuidos uniformemente en el intervalo $[0, 1]$. Nuevamente, la herramienta para informar el resultado del test es el p-valor. Valores muy bajos indican evidencia en contra de la distribución uniforme, mientras que valores no tan bajos indican que lo observado es consistente con $\mathbb{H}_0$, y por ende con H_0.

En todo experimento aleatorio que se repite una gran cantidad de veces, necesariamente ocurren cosas poco frecuentes, además de las frecuentes. Si se lanza una bola de ruleta, es poco probable que salga un doble cero, pero si se repite ese experimento cien veces, es muy probable que salga al menos un doble cero. De la misma forma, asumiendo que H_0 (y por ende $\mathbb{H}_0$) es verdadera, esperamos obtener un p-valor no muy bajo al hacer el test de Kolmogorov-Smirnov (KS), pero si realizamos muchos tests de KS a diferentes muestras, esperamos obtener algunos pocos p-valores bajos. Esto es totalmente compatible con la validez de H_0.

En el caso de este trabajo, la secuencia a testear se dividirá en bloques de igual longitud llamados *bitstreams* y los tests se realizarán sobre cada uno de estos bitstreams. De esta forma, por cada test se tendrá un p-valor por bitstream (correspondiente a realizar el test sobre dicho bitstream). Estos p-valores obtenidos son los que luego se utilizarán para realizar el test KS.

2.2. Batería de tests de Dona

Presentamos a continuación la batería de tests diseñados para este trabajo. La batería consiste de un subconjunto de los tests propuestos por Knuth más algunas variantes. De los tests propuestos por Knuth [3], no se incluyen los siguientes tests:

- **Test de Permutaciones (Permutation Test):** Este test asume que todos los símbolos de la secuencia de entrada son distintos (es decir, $|\Sigma| \geq n$. Esta es una condición que no queríamos imponer sobre nuestro input
- **Test de Rachas (Run Test):** Este test fue reemplazado en este trabajo por el test de Máxima Racha dentro de Bloque, cuya matemática es más simple.
- **Máximo de T (Maximum-of-T Test):** Test originalmente diseñado para números reales. Se decidió no incluirlo
- **Serial Correlation Test (Test de Correlación Serial):** Test originalmente diseñado para números reales, que busca en cuánta medida n_i depende de n_{i+1}. Se decidió no incluirlo

Por cada test se explicará su propósito, algunos detalles de implementación, las probabilidades esperadas y los parámetros configurables del test.

Los tests trabajan con alfabetos de tamaño arbitrario, utilizando números enteros. Dado que los alfabetos podrían estar compuestos por cualquier tipo de símbolos (como ser, por ejemplo, el alfabeto español), se establece una relación uno a uno entre el alfabeto provisto y el alfabeto canónico de igual tamaño, cuyos símbolos son los primeros números enteros no negativos. Dado

un alfabeto Σ tal que $|\Sigma| = n$, los tests van a funcionar utilizando el alfabeto canónico $\Sigma' = \{0, 1 \cdots n-1\}$. Se puede establecer un mapeo uno a uno entre ambos alfabetos.

Describimos a continuación los 8 tests de la batería. Cada uno recibe una secuencia de un alfabeto de cardinalidad arbitraria y retorna un p-valor.

2.2.1. Monobit (Test de Frecuencia)

Propósito Este test evalúa la frecuencia de cada símbolo del alfabeto Σ. El test busca determinar si la cantidad de apariciones de cada símbolo en la secuencia es aproximadamente el mismo que se esperaría para una secuencia aleatoria.

Implementación y estadístico Dada una secuencia de longitud n, la cantidad de apariciones de cada símbolo si esta fuera aleatoria sería $\frac{n}{|\Sigma|}$. El test consiste en contar la cantidad de apariciones de cada símbolo, y realizar un test χ^2 (hay una categoría por símbolo) con $|\Sigma| - 1$ grados de libertad [3].

Parámetros Este test no cuenta con parámetros adicionales.

2.2.2. Frequency Within Block (Frecuencia Dentro de Bloque)

Propósito Este test evalúa la frecuencia de cada símbolo del alfabeto $\Sigma = \{a_1, \ldots, a_n\}$ dividiendo la secuencia original en bloques iguales de largo M. Este test es una generalización del *Monobit test* (con M igual al tamaño total de la secuencia, se convierte en dicho test). Por ejemplo, si se consideran secuencias generadas de la siguiente manera:

$$\underbrace{a_1 \cdots a_1}_{M \text{ veces}} \underbrace{a_2 \cdots a_2}_{M \text{ veces}} \cdots \underbrace{a_n \cdots a_n}_{M \text{ veces}}$$

Estas secuencias pasarían el test monobit, pero fallarían este test.

Implementación y estadístico Dada una secuencia de longitud n dividida en B bloques de tamaño M, la cantidad de apariciones de cada símbolo en cada bloque si la secuencia fuera aleatoria sería $\frac{M}{|\Sigma|}$. El test consiste en contar la cantidad de apariciones de cada símbolo en cada bloque. Luego se realiza un test χ^2 (hay una categoría por cada símbolo por cada bloque, para un total de $B \cdot |\Sigma|$ categorías) con $B \cdot |\Sigma| - B$ grados de libertad (esto es porque por cada bloque hay una variable que se desprende de las demás).

Parámetros Este test cuenta con el siguiente parámetro:

- `block_size`: Tamaño de bloque a ser utilizado para dividir la secuencia (llamado M en la descripción del test) (**Default:** $n/99$, debido a que la implementación de NIST sugería utilizar no más de 99 bloques)

2.2.3. Frequency of Words (Frecuencia de Palabras)

Propósito Este test evalúa la frecuencia de todas las palabras de longitud m dentro de bloques de la secuencia original. Este test es una generalización del *Frequency Within Block* test (con m igual a 1 se convierte en dicho test). Por ejemplo, si m es igual a 3, se buscaría la frecuencia de todas las palabras de longitud 3 sobre el alfabeto Σ (si consideramos $\Sigma = \{0, 1\}$, serían las palabras $000, 001, \ldots, 110, 111$). Este test está inspirado en el Serial Test de la batería de Knuth.

Implementación y estadístico Se divide la secuencia original en B bloques de tamaño M. Luego, por cada bloque, se cuenta la cantidad de apariciones de cada palabra de longitud m (sin solapamiento).

Por ejemplo, en este bloque

$$\underbrace{010}\,\underbrace{210}\,\underbrace{011}\,\underbrace{010}\,\underbrace{012}$$

habría dos apariciones de 010, una de 210, una de 011 y una de 012.

Dada una secuencia de longitud n dividida en B bloques de tamaño M, la cantidad de apariciones de cada palabra de longitud m en cada bloque si la secuencia fuera aleatoria sería $\frac{M/m}{|\Sigma|^m}$. El test consiste en contar la cantidad de apariciones de cada palabra en cada bloque. Luego se realiza un test χ^2 (hay una categoría por cada posible palabra por cada bloque, para un total de $B \cdot |\Sigma|^m$ categorías) con $B \cdot |\Sigma|^m - B$ grados de libertad (esto es porque por cada bloque hay una variable que se desprende de las demás) [3].

Parámetros Este test cuenta con los siguientes parámetros:

- `block_size`: Tamaño de bloque a ser utilizado para dividir la secuencia (llamado M en la descripción del test) (**Default:** $n/99$).
- `pattern_length`: Tamaño de palabra a ser utilizado (llamado m en la descripción del test) (**Default:** 3).

2.2.4. Gap (Huecos)

Propósito El propósito de este test es examinar la longitud de huecos (*gaps*) entre ocurrencias de símbolos en cierto rango. Sean α y β dos números reales tales que $0 \leq \alpha < \beta \leq 1$, se busca considerar las longitudes de subsecuencias consecutivas maximales de la secuencia original en las que los elementos están fuera del intervalo $[\alpha, \beta]$. Analicemos, por ejemplo, la siguiente secuencia:

$$\underbrace{0{,}1 \quad 0{,}3 \quad 0{,}2 \quad 0{,}4 \quad 0{,}5 \quad 0{,}1 \quad 0{,}3}_{\text{gap de longitud 7}} \quad 0{,}9 \qquad \underbrace{0{,}4}_{\text{gap de longitud 1}} \qquad 0{,}7$$

Si $\alpha = 0{,}6$ y $\beta = 1{,}0$, la secuencia consiste de un *gap* de longitud 7, uno de longitud 0 y uno de longitud 1.

Implementación y estadístico Este test está originalmente diseñado para números reales, así que se propone la siguiente transformación para la entrada. Se divide la entrada en bloques de

tamaño m, considerando cada tira de m números como si se tratase de un número real de la forma $0, m$ (escrito en base $|\Sigma|$). Por ejemplo, supongamos que $m = 3$ y la secuencia es la siguiente:

$$1 \quad 1 \quad 2 \quad 1 \quad 3 \quad 1$$

En este caso, obtendríamos esta nueva secuencia:

$$0{,}112 \quad 0{,}131$$

Se convierte esta nueva secuencia a números para hacer las cuentas. Si, por ejemplo, el alfabeto del ejemplo fuera $\Sigma = \{0, 1, 2, 3\}$, se interpretaría a 0,112 y 0,131 como números escritos en base 4.

Una vez transformada la entrada, se procede a ejecutar el algoritmo definido en el libro de Knuth. ([3], página 60) Se cuenta la cantidad de gaps de longitudes $1 \cdots t - 1$ y de longitud $\geq t$.

Nota: Si el valor de t no es provisto, se busca uno de forma tal que no queden categorías vacías hacia el final del histograma.

Las probabilidades se calculan de la siguiente manera [3], donde $p = \beta - \alpha$:

- Para i tal que $0 \leq i < t$:

$$p_i = p(1 - p)^i$$

- Para $i = t$:

$$p_i = (1 - p)^i$$

Luego se realiza un test χ^2 (hay $t + 1$ categorías, una por cada longitud menor a t y una por longitud $\geq t$) con t grados de libertad.

Parámetros Este test cuenta con los siguientes parámetros:

- `alpha`: Extremo inferior del intervalo a considerar (llamado α en la descripción del test) (**Default:** $1/3$).
- `beta`: Extremo superior del intervalo a considerar (llamado β en la descripción del test) (**Default:** $2/3$).
- `m`: Cantidad de números a tomar de la secuencia original para realizar la transformación a números reales (**Default:** 5).
- `t`: Última categoría del histograma.

2.2.5. Poker

Propósito Este test considera la entrada dividida en secuencias (o "manos") de 5 símbolos sin solapamiento y observar cuál de los siguientes patrones corresponde a cada mano.

- Todas diferentes: *abcde*

- Par: *aabcd*
- Doble Par: *aabbc*
- Pierna: *aaabc*
- Full: *aaabb*
- Poker: *aaaab*
- Cinco Iguales: *aaaaa*

Luego sobre la cantidad de manos de cada tipo se puede realizar un test de χ^2.

Knuth propone una versión simplificada de este test en el que únicamente se cuenta la cantidad de símbolos distintos que hay en cada mano, quedando las siguientes categorías:

- 5 diferentes: Todas Diferentes
- 4 diferentes: Par
- 3 diferentes: Doble Par y Pierna
- 2 diferentes: Full y Poker
- 1 diferentes: Cinco Iguales

Implementación y estadístico Para la implementación de este test se utilizan los números de Stirling de segundo orden, que se definen a continuación:

Los números de Stirling de segundo orden representan el número de formas de hacer una partición de un conjunto de n objetos en k clases disjuntas no vacías. El número de Stirling de segundo orden n, k se nota $\left\{ {n \atop k} \right\}$ y se calcula de la siguiente manera:

$$\left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^n$$

Este test se generaliza a manos de tamaño k. Consideramos entonces secuencias de tamaño k que contienen r valores diferentes. La probabilidad de que haya r valores distintos se calcula de la siguiente manera [3]:

$$p_r = \frac{|\Sigma|(|\Sigma|-1) \cdot \ldots \cdot (|\Sigma|-r+1)}{|\Sigma|^k} \left\{ {k \atop r} \right\}$$

Como p_r suele ser muy pequeña para valores muy chicos de r se agrupan algunas categorías juntas para que no queden vacías luego. Esto se hace agrupando las categorías a izquierda y a derecha de forma tal que la primer y la última categoría tengan una probabilidad p tal que, de haber m observaciones (manos) en total, $p \cdot m$ sea mayor o igual a un número determinado (5 por default).

Luego se realiza un test χ^2 (hay tantas categorías como cantidad de valores diferentes posibles en una mano).

Parámetros Este test cuenta con el siguiente parámetro:

- `hand_size`: Tamaño de la mano (llamado k en la descripción del test) (**Default:** 5)

2.2.6. Coupon Collector (Álbum de Figuritas)

Propósito Este test ve la secuencia como una tira de figuritas. La idea es contar cuántos símbolos hay que ver hasta dar con un set completo del alfabeto (cuántas figuritas comprar hasta llenar el álbum). Por ejemplo, considerar la siguiente secuencia del alfabeto $\Sigma = \{0, 1, 2, 3\}$

$$\underbrace{003102}_{\text{Primer album}} \quad \underbrace{22301}_{\text{Segundo album}}$$

En este caso las longitudes observadas hasta tener un set completo son 6 y 5 respectivamente.

Implementación y estadístico Se toma la entrada y se recorre midiendo las longitudes de cada álbum completo. Se cuenta la cantidad de álbumes de longitud $|\Sigma|, \ldots, t-1$ y de longitud $\geq t$.

Nota: Llamamos $t-1$ al máximo tamaño de album más grande que queremos identificar propiamente. Los álbumes de mayor tamaño estarán en la categoría de tamaño mayor o igual a t. Si el valor de t no es provisto, se busca uno de forma tal que no queden categorías vacías hacia el final del histograma.

Dado el valor t, la probabilidad de que un álbum tenga longitud r se calcula de la siguiente manera [3] (se utilizan números de Stirling de segundo orden, previamente definidos en la sección 2.2.5):

- Para r tal que $|\Sigma| \leq r < t$:

$$p_r = \frac{|\Sigma|!}{|\Sigma|^r} \left\{ {r-1 \atop d-1} \right\}$$

- Para $r = t$:

$$p_r = \frac{|\Sigma|!}{|\Sigma|^{t-1}} \left\{ {t-1 \atop d} \right\}$$

Luego se realiza un tet χ^2 (hay $t - |\Sigma| + 1$ categorías, una por cada longitud entre $|\Sigma|$ y $t-1$ y una $\geq t$) con $t - |\Sigma|$ grados de libertad.

Parámetros Este test cuenta con el siguientes parámetro:

- `t`: Última categoría del histograma.

2.2.7. Longest Run Within Block (Racha más Larga dentro de Bloque)

Propósito Una racha es una secuencia maximal de símbolos idénticos consecutivos. El test de longitud de la racha más larga dentro de un bloque consiste en elegir un símbolo (por ejemplo 0), dividir la entrada en bloques contiguos no superpuestos de igual longitud M, y dentro de cada

uno de estos bloques se computa la longitud de la racha más larga de dicho símbolo. Notar que en un bloque de longitud M puede haber más de una racha, y dentro de esas hay que tomar la longitud de la más larga. Notar también que el test mira únicamente un símbolo. Podría correrse para cada símbolo del alfabeto. El test determina si las longitudes encontradas se corresponden con lo esperado por aleatoriedad.

Implementación y estadístico Fijado un símbolo t, llamamos $D(n,k)$ a la cantidad de secuencias de longitud n cuya racha más larga de ts tiene longitud k (y a su vez, la secuencia no comienza y termina con t). Para simplificar, asumimos $k \leq n-2$. Dada una secuencia que tiene una racha más larga de longitud k, consideramos verla como tres segmentos consecutivos. Un primer segmento que no empieza ni termina en t y cuya racha de ts es menor que k. Un segundo segmento que es exactamente la racha de ts de longitud k. Por último el segmento que no empieza ni termina en t y cuya racha de ts es menor o igual que k. Tanto el primer como el tercer segmento podrían ser vacíos.

$$\underbrace{\cdots j}_{j \neq t}\ \underbrace{t\cdots t}_{\text{longitud } k}\ \underbrace{i\cdots}_{i \neq t}$$

Por ejemplo, con $t = 0$ y $\Sigma = \{0,1,2,3\}$:

$$\underbrace{}_{\substack{\text{primer bloque}\\ \text{vacío}}}\ \underbrace{0000}_{k=4}\ \underbrace{12000312}_{\substack{\text{la racha de 0}\\ \text{es de longitud <k}}}$$

Se cumple la siguiente relación recursiva:

$$D(n,k) = \sum_{j=2}^{n-k-1} \sum_{s=0}^{k-1} D(j-1,s) \sum_{t=0}^{k} D(n-j+1-k,t)$$

donde j indexa la primera aparición de la racha, s indexa la racha del primer segmento, y t indexa la racha del tercer segmento. Y para todo m, $D(m,0) = (|\Sigma|-1)^m$, $D(m,m-2) = (|\Sigma|-1)^2$. Además, se tiene que para todo k, $D(1,k) = (|\Sigma|-1)$ y $D(2,k) = (|\Sigma|-1)^2$.

Sea $C(n,k)$ la cantidad de cadenas de longitud n cuya racha de ts tiene longitud k. Es fácil ver que

$$C(n,k) = D(n+2,k)/(|\Sigma|-1)^2$$

A partir de $C(n,k)$ determinamos la probabilidad p_k de que en una secuencia de n símbolos haya una racha de k ts, para $k = 1,2,\ldots n-2$.

Dada una secuencia de símbolos del alfabeto Σ, se la divide en bloques de tamaño n. Por cada bloque se registra la longitud de la corrida más larga de t. De esta forma se cuentan la cantidad de corridas de longitud k para k desde 0 hasta la máxima longitud de racha de t que aparezca en algún bloque (vamos a llamarla r).

También se calculan las probabilidades teóricas p_k para $k = 1,2,\ldots r-1$ y una última categoría para las rachas de longitud mayor o igual que r. Esta es la distribución teórica esperada para bloques de longitud 400 en el alfabeto que contiene los números del 0 al 9:

k	p_k
1	$4{,}977414122938492e-19$
2	0,025339385657741895
3	0,6727353262392227
4	0,26680252353924366
5	0,03156393698492928
6	0,003203288734464395
7	0,0003200694497359544
8	$3{,}1931400653089694e-05$
9	$3{,}185094068318768e-06$
10	$3{,}176999412983973e-07$

Por último, se agrupan algunas categorías de los extremos para evitar que queden categorías vacías en el test, y se efectúa un test χ^2 con los valores observados.

Parámetros Este test cuenta con los siguientes parámetros:

- `block_size`: Tamaño de bloque (**Default:** 40).
- `character`: Símbolo a ser tomado en cuenta para el test (llamado t en la descripción del test) (**Default:** 0).

2.2.8. Collision (Colisión)

Propósito Este test se parece a evaluar una función de hashing. Se supone que se tienen u urnas y n bolas, donde u es mucho mayor que n. Es de esperarse que la mayoría de las bolas caigan en urnas previamente vacías. Si una bola cae en una urna ocupada, se dice que hay una colisión. El test cuenta la cantidad de colisiones y se fija que no sean demasiadas ni muy pocas.

Implementación y estadístico Para implementar este test, se agrupa la entrada en segmentos de tamaño m. De esta forma, dado un alfabeto de tamaño $|\Sigma|$, se pasa a tener un alfabeto de tamaño $|\Sigma|^m$, y en lugar de tener n elementos en la secuencia, se pasa a tener $\frac{n}{m}$. De esta manera se puede obtener la condición del test que requiere que haya muchas más urnas que bolas. Cada urna se corresponde a una tira de m elementos del alfabeto Σ. Por ejemplo, si el alfabeto es $\Sigma = \{0, 1, 2, 3\}$ y la secuencia original es

$$023123120310$$

Si $m = 4$, la secuencia con la que se va a realizar el test es la siguiente:

$$\underbrace{0231}\underbrace{2312}\underbrace{0310}$$

Knuth da las probabilidades de que ocurran c colisiones (que es la probabilidad de que $\frac{n}{m} - c$ urnas estén ocupadas [3]). Esto es (se utilizan números de Stirling de segundo orden, previamente definidos en la sección 2.2.5):

$$p_c = \frac{|\Sigma|^m(|\Sigma|^m - 1) \cdot \ldots \cdot (|\Sigma|^m - \frac{n}{m} + c + 1)}{(|\Sigma|^m)^2} \left\{ \begin{matrix} \frac{n}{m} \\ \frac{n}{m} - c \end{matrix} \right\}$$

Knuth utiliza estas probabilidades para calcular puntos percentuales y generar una tabla como la siguiente:

colisiones $\leq$	101	108	119	126	134	145	153
con probabilidad	,009	,043	,244	,476	,742	,946	,989

Esto significa que con 0,009 % de probabilidad hay menos de 101 colisiones, con 0,244 % de probabilidad hay menos de 119, etc.

Los puntos elegidos por Knuth para llegar a estos cálculos son 0,01, 0,05, 0,25, 0,50, 0,75, 0,95, 0,99, 1,00.

Knuth no elabora sobre cómo evaluar el test una vez obtenida esta tabla. Se optó por seguir la sugerencia mencionada en [4]. La propuesta consiste en utilizar esa tabla para generar categorías para un test χ^2. Por lo tanto, después de transformar la entrada, se la divide en bloques de tamaño B. Para cada uno de estos bloques se cuenta la cantidad de colisiones y se cuenta cuántas observaciones cayeron en cada categoría. Se realiza luego el test χ^2, utilizando las categorías definidas previamente.

Parámetros Este test cuenta con los siguientes parámetros:

- `m`: Cantidad de números a tomar de la secuencia original para realizar la transformación (**Default:** 5).
- `block_size`: Tamaño de bloque (llamado B en la descripción del test) (**Default:** 1000).
- `percentage_points`: Puntos sobre los cuales tomar los porcentajes (**Default:** [0.01, 0.05, 0.25, 0.50, 0.75, 0.95, 0.99, 1.00]).

Nota sobre estos parámetros: Se debe tener cuidado con que después de la transformación de la entrada original no queden muy pocos números. Esto podría causar que luego queden pocos bloques y las categorías del test χ^2 queden vacías.

2.3. Guía de Uso

2.3.1. Requerimientos

La batería se encuentra implementada en Python 3. Para correrla es necesario utilizar una versión de Python mayor o igual a 3.8 y tener instalado pip3.

El código se encuentra en el siguiente repositorio en github:

```
https://github.com/NDonatucci/donut_tests
```

Se incluye un archivo de requerimientos para python. Para instalarlos:

```
pip3 install requirements.txt
```

2.3.2. Implementación

La batería consiste en un archivo de python principal. En este se leen las configuraciones y se procede luego a ejecutar cada test y luego reportar sus resultados.

Cada test está implementado en su propio archivo, y consiste de una función que recibe como parámetros:

- La secuencia a testear
- El tamaño del alfabeto utilizado
- Los parámetros del test, en forma de diccionario
- El nivel de significancia del test

Cada test devuelve una tupla con un valor de verdad que indica si el test es éxito o fallo, y el p-valor asociado.

Hay además un archivo *results.py* que contiene la lógica de generación de reportes de resultados y un archigo *utils.py* con funciones que reusan varios tests.

Para extender la batería agregando un nuevo test sólo se debe agregar un archivo con el test nuevo implementado.

2.3.3. Configuración

La batería utiliza un archivo de configuración en formato $json$. Contiene configuraciones globales y configuraciones por test. El archivo debe llamarse *config.json*.

Configuraciones globales

- **tets:** Una lista con los nombres de los tests que quieren correrse
- **significance_level:** Nivel de significancia para los tests.

Configuraciones por test

- **configs:** Por cada test se pueden configurar los parámetros descriptos en la sección *parámetros* de cada test.

Ejemplo

```
{
  "tests": [
    "gap"
  ],
  "significance_level": 0.01,
  "configs": {
    "frequency_within_block": {
    },
    "frequency_of_words": {
```

```
      "pattern_length": 3
    },
    "gap": {
      "alpha": 0.3,
      "beta": 0.6,
      "m":20
    },
    "longest_run_within_block": {
      "character": 0
    },
    "poker": {
      "hand_size": 2
    },
    "collision": {
      "m": 5,
      "block_size": 1000
    },
    "monobit": {
    },
    "coupon_collector": {
    }
  }
}
```

Se incluye un ejemplo con el código. (archivo llamado *config.json*)

2.3.4. Input

El input esperado consiste de un archivo *csv* que contenga la secuencia entera, con exactamente un número por línea. Se incluyen ejemplos con el código (por ejemplo, *demo_random.csv*).

2.3.5. Uso

La batería se corre como un script python desde la línea de comando. Posee dos parámetros posicionales:

Parámetros

- `alphabet_size`: Tamaño del alfabeto a utilizar. Es un número entero. (parámetro requerido).
- `filename`: Archivo donde encontrar los datos para el input. (parámetro requerido).

Luego, se tienen otros parámetros opcionales:

- `stream_size`: Tamaño del bitstream (**Default:** 10000).
- `config`: Nombre del archivo de configuración (**Default:** *config.json*).

Ejemplos Un ejemplo posible de uso, con los parámetros opcionales default y un alfabeto de dos símbolos:

```
python3 donut_tests.py 2 file.csv
```

Un ejemplo más, utilizando los parámetros opcionales para definir tamaño de bitstream en cien mil, alfabeto de diez símbolos y archivo de configuración *conf.json*:

```
python3 donut_tests.py 2 file.csv --stream_size 100000 --config conf.json
```

2.3.6. Reporte de resultados

Cuando se corre la batería sobre un archivo, se imprime un resumen en pantalla que incluye el nombre de cada test, el resultado del Kolmogorov-Smirnov y `PASS` o `FAIL` dependiendo el resultado.

Además, se genera una carpeta *out* que contiene el output de la corrida. Dentro de esta carpeta se encontrará una carpeta por cada test que se corrió. (por ejemplo, en *out/monobit* se encontrarán los resultados del test monobit).

Dentro de cada carpeta se encontrarán dos archivos: *test.result* y *test.png*

test.result Este es el archivo que contiene el resultado de la corrida. Es un archivo que contiene una línea por cada bitstream indicando el p-valor obtenido y `PASS` o `FAIL` dependiendo de si pasó o no el test. Debajo de todas estas líneas se incluye una línea mas con el p-valor obtenido en el test Kolmogorov-Smirnov.

Por ejemplo:

```
0.6084334365843856 PASS
0.4225759836091928 PASS
0.4480943793728133 PASS
0.36457820271914543 PASS
0.19957804088486955 PASS
0.7934887861657944 PASS
0.7818854805743349 PASS
0.008407354562127406 FAIL
...
0.23086624356885949 PASS
0.4235535019791607 PASS
0.7194398190544982 PASS
0.3116008635025613 PASS
0.3551769054161957 PASS
0.3984875461892538 PASS

----------
KS VALUE: 0.8135035898839948
```

test.png Se incluye una imagen del histograma que representa la distribución de los p-valores obtenidos en cada bit-stream, utilizando diez buckets.

Por ejemplo:

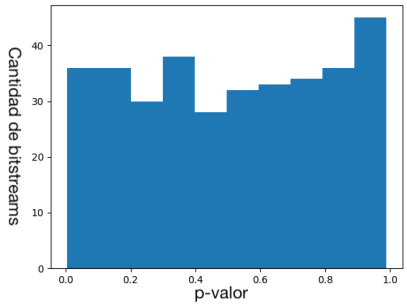

Fig. 2.1: Ejemplo de histograma

2.4. Casos de Estudio

Abusaremos del lenguaje y diremos que *convertimos una secuencia de base n a base m* cuando se parte de una secuencia de un alfabeto Σ tal que $|\Sigma| = n$, interpretamos la secuencia original como un número entero utilizando su representación en base n y a este número lo pasamos a base m. Luego se interpreta ese número como una secuencia en el alfabeto Σ' tal que $|\Sigma'| = m$.

Se corrió la batería sobre diferentes secuencias para analizar los resultados obtenidos, utilizando siempre tres tamaños de bit stream. En primer lugar, usando un solo bit stream que consiste de la secuencia entera. Luego, utilizando bit streams de tamaño 1 millón, y por último, utilizando bit streams de tamaño 100 mil. Los símbolos que sobran al final de la secuencia se tiran (en los casos de bistreams de tamaño 1 millón y cien mil).

Nota: En el test de colisión, hay algunas posiciones en las tablas de resultados que aparecen como N/A. Por los parámetros utilizados, cuando se tenían bit streams de tamaño 100 mil en base 2, quedaban muy pocos bloques, por lo tanto quedaban categorías vacías en el test χ^2. Por lo tanto, se saltean los resultados para esas instancias.

El alfabeto es $\Sigma = \{0, 1\}$ para las secuencias en base 2 y $\Sigma' = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ para las de base 10.

Se pensaron los casos con el propósito de encontrar secuencias que cambien sus propiedades al realizar pasaje a base 2. Se intentó generar secuencias que fallarían los tests en base 10 y se quería ver si los pasaban en base 2. En las siguientes secciones se describen las secuencias utilizadas y los resultados obtenidos.

2.4.1. Fibonacci

Originalmente partimos de Fibonacci como una secuencia claramente no aleatoria para empezar a realizar nuestras pruebas.

Fibonacci en base 2 Esta secuencia consiste en los primeros diez mil números de la secuencia de Fibonacci convertidos a base dos uno a uno y luego concatenados.

Se corrió la batería utilizando tres tamaños distintos de bit stream.

	Stream size = 34.708.959	Stream size = 1.000.000	Stream size = 100.000
Monobit	PASS	PASS	PASS
Frequency Within Block	PASS	PASS	PASS
Frequency of Words	PASS	PASS	PASS
Gap	PASS	PASS	PASS
Poker	PASS	PASS	PASS
Coupon Collector	PASS	PASS	PASS
Longest Run Within Block	PASS	PASS	PASS
Collision	PASS	PASS	N/A

Fibonacci en base 10 Esta secuencia consiste en los primeros diez mil números de la secuencia de Fibonacci concatenados.

Se corrió la batería utilizando tres tamaños distintos de bit stream.

	Stream size = 10.451.934	Stream size = 1.000.000	Stream size = 100.100
Monobit	PASS	PASS	PASS
Frequency Within Block	PASS	PASS	PASS
Frequency of Words	PASS	PASS	PASS
Gap	PASS	PASS	PASS
Poker	PASS	PASS	PASS
Coupon Collector	PASS	PASS	PASS
Longest Run Within Block	PASS	PASS	PASS
Collision	PASS	PASS	PASS

Nuestros resultados son consistentes con los que reportan Cangiotti y Taufer [1]. Por otro lado la secuencia de Fibonacci modulo m tiene período [8] y [2].

Fibonacci en base 10 convertida a base 2 Esta secuencia consiste en convertir la secuencia de la sección anterior a base 2 de la forma descripta previamente.

Se corrió la batería utilizando tres tamaños distintos de bit stream.

	Stream size = 34.720.571	Stream size = 1.000.000	Stream size = 100.000
Monobit	PASS	PASS	PASS
Frequency Within Block	PASS	PASS	PASS
Frequency of Words	FAIL	PASS	PASS
Gap	PASS	PASS	PASS
Poker	PASS	PASS	PASS
Coupon Collector	PASS	PASS	PASS
Longest Run Within Block	PASS	PASS	FAIL
Collision	PASS	PASS	N/A

Vale comentar que la secuencia de Fibonacci sobre alfabeto $\{a, b\}$ definida mediante la concatenación en vez de la suma falla el test de Monobit, ver Remark en Seccion 2.3.1 [7].

2.4.2. Todas las secuencias de longitud hasta N

Secuencias de longitud hasta 6 en base 10 Esta secuencia consiste en concatenar todas las secuencias de longitud 1 hasta 6. Por ejemplo:

$$0 \quad 1 \quad \cdots \quad 9 \quad 00 \quad 01 \quad \cdots \quad 99 \quad 000 \quad 001 \quad \cdots \quad 999999$$

La intuición detrás de esta secuencia es que en base 10 no debería pasar ningún test, o a lo sumo el monobit (corrido sobre toda la secuencia, no sobre un bloque). Buscabamos ver qué sucedería con ellos después de la conversión.

Se corrió la batería utilizando tres tamaños distintos de bit stream.

	Stream size = 6.543.210	Stream size = 1.000.000	Stream size = 100.000
Monobit	PASS	FAIL	FAIL
Frequency Within Block	FAIL	FAIL	FAIL
Frequency of Words	FAIL	FAIL	FAIL
Gap	FAIL	FAIL	FAIL
Poker	FAIL	FAIL	FAIL
Coupon Collector	FAIL	FAIL	FAIL
Longest Run Within Block	FAIL	FAIL	FAIL
Collision	FAIL	FAIL	FAIL

Tiras de longitud hasta 6 en base 10 convertida a base 2 Esta secuencia consiste en convertir la secuencia de la sección anterior a base 2 de la forma descripta previamente.

Se corrió la batería utilizando tres tamaños distintos de bit stream.

	Stream size = 21.736.067	Stream size = 1.000.000	Stream size = 100.000
Monobit	PASS	FAIL	FAIL
Frequency Within Block	PASS	PASS	PASS
Frequency of Words	PASS	PASS	PASS
Gap	PASS	PASS	PASS
Poker	PASS	FAIL	FAIL
Coupon Collector	PASS	PASS	PASS
Longest Run Within Block	PASS	PASS	FAIL
Collision	PASS	PASS	N/A

2.4.3. Secuencia alterada

Secuencia alterada en base 10 Esta secuencia se generó partiendo de una secuencia de 7.000.000 números entre 0 y 9 generada de manera pseudoaleatoria. Luego, por cada símbolo 1 presente en la secuencia, con una probabilidad de 10 % se cambia por un 0. Por lo tanto, resulta

en una secuencia con números del dos al nueve distribuidos de manera pseudoaleatoria, y una distribución de unos y ceros alterada.

La intuición detrás de esta secuencia es que, al tener esta alteración, los tests de frecuencia fallarían todos. Buscabamos ver qué sucedería con estos luego de la conversión.

	Stream size = 7.000.000	Stream size = 1.000.000	Stream size = 100.000
Monobit	FAIL	FAIL	FAIL
Frequency Within Block	FAIL	FAIL	FAIL
Frequency of Words	FAIL	FAIL	FAIL
Gap	PASS	PASS	PASS
Poker	PASS	PASS	PASS
Coupon Collector	PASS	PASS	PASS
Longest Run Within Block	FAIL	FAIL	FAIL
Collision	PASS	PASS	PASS

Secuencia alterada en base 10 convertida a base 2 Esta secuencia consiste en convertir la secuencia de la sección anterior a base 2 de la forma descripta previamente.

Se corrió la batería utilizando tres tamaños distintos de bit stream.

	Stream size = 23.253.497	Stream size = 1.000.000	Stream size = 100.000
Monobit	PASS	PASS	PASS
Frequency Within Block	PASS	PASS	PASS
Frequency of Words	PASS	PASS	PASS
Gap	PASS	PASS	PASS
Poker	PASS	PASS	PASS
Coupon Collector	PASS	PASS	PASS
Longest Run Within Block	PASS	PASS	FAIL
Collision	PASS	PASS	N/A

2.4.4. Secuencia aleatoria

Secuencia aleatoria en base 10 Esta secuencia se generó utilizando el generador de números aleatorios de Python de *numpy*, utilizando la semilla 1994.

	Stream size = 10.000.000	Stream size = 1.000.000	Stream size = 100.000
Monobit	PASS	PASS	PASS
Frequency Within Block	PASS	PASS	PASS
Frequency of Words	PASS	PASS	PASS
Gap	PASS	PASS	PASS
Poker	PASS	PASS	PASS
Coupon Collector	PASS	PASS	PASS
Longest Run Within Block	PASS	PASS	PASS
Collision	PASS	PASS	PASS

2.5. Resumen de Resultados y Conclusiones

En la sección anterior presentamos ejemplos concretos donde transformar una secuencia a un alfabeto de dos símbolos traía resultados no fiables. Resumimos los resultados en la siguiente tabla (incluímos para cada una de las secuencias en base 10 cuántos tests pasaron en cada tamaño distinto de bitstream y al lado la misma información para su versión convertida a base 2):

	Original en Base 10	**Transformación a Base 2**
Fibonacci	8/8 **Bitstream:** toda la secuencia 8/8 **Bitstream:** 1.000.000 8/8 **Bitstream:** 100.000	7/8 **Bitstream:** toda la secuencia 8/8 **Bitstream:** 1.000.000 6/7 **Bitstream:** 100.000
Todas las secuencias de longitud hasta N	1/8 **Bitstream:** toda la secuencia 0/8 **Bitstream:** 1.000.000 0/8 **Bitstream:** 100.000	8/8 **Bitstream:** toda la secuencia 6/8 **Bitstream:** 1.000.000 4/7 **Bitstream:** 100.000
Secuencia alterada	4/8 **Bitstream:** toda la secuencia 4/8 **Bitstream:** 1.000.000 4/8 **Bitstream:** 100.000	8/8 **Bitstream:** toda la secuencia 8/8 **Bitstream:** 1.000.000 6/7 **Bitstream:** 100.000

Como fue mencionado previamente, el propósito de estos casos era encontrar secuencias donde la transformación propuesta resulte en resultados muy distintos.

Comenzamos eligiendo Fibonacci como secuencia evidentemente no aleatoria, pero resultó pasar todos los tests en base 10, para nuestra sorpresa. Esto es consistente con lo reportado por Cangiotti y Taufer [1]. Tuvo resultados parecidos una vez transformada.

La secuencia de *Todas las Secuencias de Longitud Hasta N*, presentada en la sección 2.4.2, falla todos los tests salvo el de frecuencia básica en base 10 (únicamente si se toma toda la secuencia como único bitstream). Esto es consistente con la idea con la que fue creada. Su transformación a un alfabeto de dos símbolos, pasó todos los tests utilizando un único bistream. Utilizando varios bitstreams más pequeños también pasó muchos más tests que la secuencia original (que no pasaba ninguno).

La *Secuencia Alterada*, presentada en la sección 2.4.3, es una secuencia diseñada para fallar los tests que se basan en frecuencia. Observamos que falla los tres tests de frecuencia (monobit, frequency within block y frequency of words) y el test de longest run within block (porque se realizó observando el símbolo 0, ¡cuya frecuencia estaba alterada!). La transformación de la secuencia Alterada a un alfabeto de dos símbolos pasa todos los tests de frecuencia.

Podemos ver, entonces, que transformar nuestra entrada para utilizar una batería ya existente puede ser no fiable, es decir, podríamos estar obteniendo resultados falsos. Por lo tanto, es importante poder contar con tests que acepten alfabetos arbitrarios

Se pone entonces a disposición de quien quiera utilizarla y/o extenderla una batería de tests de aleatoriedad con tests adaptados a alfabetos mas grandes que el $\{0, 1\}$.

2.6. Trabajo Futuro

Como trabajo futuro podría extenderse la batería con más tests. En principio, adaptar los tests de Knuth que han quedado afuera, así como también tests más complejos de otras baterías, que nos permitan, por ejemplo, obtener un resultado donde fibonacci falle.

Podría también mejorarse el tiempo de cómputo precalculando resultados para números de stirling (o de la función D del test de longitud de racha mas larga dentro de bloque) para las bases más comunes.

Podría también mejorarse el reporte de resultados. Por ejemplo, utilizar gráficos más expresivos como el boxplot o brindar más información de la ya presentada.

También podría evaluarse la viabilidad de hacer una aplicación online para no obligar a los usuarios a instalarse python.

BIBLIOGRAFÍA

[1] Nicolo Cangiotti and Daniele Taufer. Normal and pseudonormal numbers, 2021. https://arxiv.org/pdf/2102.05925.pdf.

[2] Friedrich Gebhardt. Generating pseudo-random numbers by shuffling a Fibonacci sequence. volume 21, pages 708–709. 1967.

[3] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981.

[4] Onur Koçak, Fatih Sulak, Ali Doganaksoy, and Muhiddin Uğuz. Modifications of knuth randomness tests for integer and binary sequences. *Commun. Fac. Sci. Univ. Ank. Ser. A1 Math. Stat.*, 67(2):64–81, 2018.

[5] George Marsaglia. The marsaglia random number cdrom including the diehard battery of tests of randomness. `https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/`, 1995. Florida State University.

[6] National Institute of Standards and US Department of Commerce Technology, Technology Administration. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Special Publication 800-22 - Revision 1a, Lawrence E Bassham III - Revised April 2010.

[7] Michel Waldschmidt. Words and transcendence. In *Analytic Number Theory - Essays in Honour of Klaus Roth*, chapter 31, pages 449–470. Cambridge University Press, Cambridge, 2009.

[8] D. D. Wall. Fibonacci series modulo m. *The American Mathematical Monthly*, 67(6):525–532, 1960.